Crosscutting, what is and what is not?: A Formal definition based on a Crosscutting Pattern

Technical Report TR28/07. University of Extremadura

José María Conejero
Quercus SEG,
University of Extremadura,
Avda. de la Universidad s/n,
10071, Cáceres, Spain
chemacm@unex.es

Juan Hernández
Quercus SEG,
University of Extremadura,
Avda. de la Universidad s/n,
10071, Cáceres, Spain
juanher@unex.es

Elena Jurado
University of Extremadura,
Avda. de la Universidad s/n,
10071, Cáceres, Spain
elenajur@unex.es

Klaas van den Berg
Software Engineering Group
University of Twente,
P.O. Box 217, 7500 AE
Enschede, the Netherlands
k.g.vandenberg@ewi.utwente.nl

ABSTRACT
Crosscutting is usually described in terms of scattering and tangling. However, the distinction between these concepts is vague, which could lead to ambiguous statements. Sometimes, precise definitions are required, e.g. for the formal identification of crosscutting concerns. We propose a conceptual framework for formalizing these concepts based on a crosscutting pattern that shows the mapping between elements at two levels, e.g. concerns and representations of concerns. The definitions of the concepts are formalized in terms of linear algebra, and visualized with matrices and matrix operations. In this way, crosscutting can be clearly distinguished from scattering and tangling. Using linear algebra, we demonstrate that our definition generalizes other definitions of crosscutting as described by Masuhara & Kiczales [21] and Tonella and Ceccato [28]. The framework can be applied across several refinement levels assuring traceability of crosscutting concerns. Usability of the framework is illustrated by means of applying it to several areas such as change impact analysis, identification of crosscutting at early phases of software development and in the area of model driven software development.

Keywords
Aspect-Oriented Software Development, Scattering, Tangling, Crosscutting, Crosscutting Concerns

1. INTRODUCTION

One of the key principles in Aspect-Oriented Software Development (AOSD) is Separation of Concerns (SOC) [12]. A concern can be defined very generally as an item in an engineering process about which it cares [9]. Related with this principle is the problem of crosscutting concerns. Crosscutting is usually described in terms of scattering and tangling, e.g. crosscutting is the scattering and tangling of concerns arising due to poor support for their modularization. However, the distinction between these concepts is vague, sometimes leading to ambiguous statements and confusion, as stated in [16]:

.. the term "crosscutting concerns" is often misused in two ways: To talk about a single concern, and to talk about concerns rather than representations of concerns. Consider "synchronization is a crosscutting concern": we don't know that synchronization is crosscutting unless we know what it crosscuts. And there may be representations of the concerns involved that are not crosscutting.

The goal of this paper is to propose a conceptual framework where consistent and precise definitions of scattering, tangling and crosscutting are provided. A precise definition of crosscutting is mandatory for the identification of crosscutting concerns at any phase of the software life cycle. The focus is not on specific examples although they should fit in this general framework. The description of crosscutting presented here is similar to other definitions of Masuhara & Kiczales [21] and of Tonella and Ceccato

[28] [8]. A formal comparison of these definitions and ours is shown. We demonstrate that our definition generalizes the aforementioned ones.

Furthermore we show the applicability of our conceptual framework for the identification and traceability of crosscutting concerns across software development phases. Usability of the framework is also shown in other areas such as the definition of an aspect oriented metrics suite. We show how the framework allows the representation of existing metrics defined in [25] and [10] and how to extend such metrics with new ones.

The paper is structured as follows. In Section 2, we introduce our formal definition of crosscutting, tangling and scattering based on the crosscutting pattern and compare it with other definitions. In Section 3, we describe how to represent and visualize crosscutting by means of dependence graphs and matrices. We show some matrix operations designed to identify crosscutting and some real examples where we distinguish between scattering, tangling and crosscutting. In Section 4, we discuss the cascading of crosscutting patterns which can be used for traceability analysis. In Section 5 we show how to use the framework to assess the degree of crosscutting in a system. Finally in Sections 6 and 7, we present related work and conclusions of the paper.

2. DEFINITIONS OF CROSSCUTTING

In this section we focus on our formal definition of crosscutting based on a crosscutting pattern. Other definitions have been presented in the literature such as [21] and [28] [8]. A formal comparison of definitions is shown at the end of section.

2.1 Definition based on Crosscutting Pattern

In this section, we first introduce an intuitive notion of crosscutting, which will be generalized in a crosscutting pattern. Based on this pattern, we provide precise definitions of scattering, tangling and crosscutting and their relation.

For example, assume we have three concerns shown as elements of a source in Figure 1, and four requirements (e.g. viewpoints or use cases) shown as elements of a target.

This picture is consistent with the quotation in the Introduction. Intuitively, we could say that *s1* crosscuts *s3* for the given relation between source and target elements. In this figure, we only show two abstraction levels. Multiple intermediate levels between source and target may exist. In the following section, we generalize this intuition by means of a crosscutting pattern. Furthermore, we focus on definitions of crosscutting, tangling and scattering.

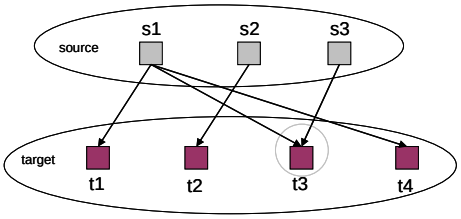

Figure 1. Trace relations between source and target elements

2.1.1 Crosscutting pattern

Our proposition is that crosscutting can only be defined in terms of *'one thing' with respect to 'another thing'*. Accordingly and from a mathematical point of view, what this means is that we have two domains related to each other through a mapping. We use here the general terms *source* and *target* (as in [22]) to denote these two domains and the trace relationship is the mapping relating these domains (Figure 2).

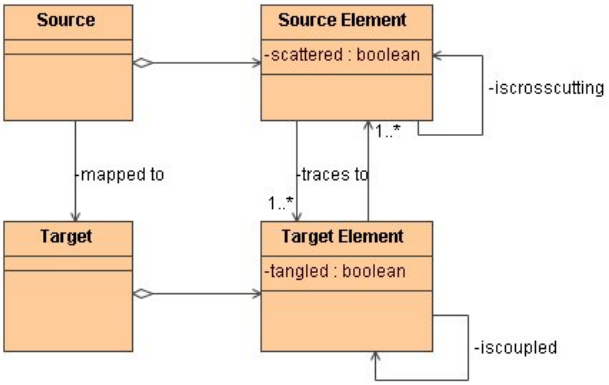

Figure 2. Crosscutting pattern

We use the term of Crosscutting Pattern to denote the situation where source and target are related to each other through trace dependencies. We use the term *pattern* as in design patterns [14], in the sense of being a general description of frequently encountered situations [21], [28]. In the Crosscutting Pattern, the mappings between source and target elements are captured in trace dependency relationships. In Figure 3, we show a model of these relationships. Ramesh and Jarke [23] show a more detailed model about traceability where these and other more specific relations are explained. The UML 2.0 specification [27] also covers such relationships. In [18] the authors show other taxonomy of traceability relationships. The model shown in Figure 3 is based on the previous ones covering some important trace relationships of interest for crosscutting identification.

As shown in Figure 3 we focus just on the following types of trace relationships: refinement, elaboration, evolution and representation. These relationships may be applied to different domains where we can find them. For example:

- *Refinement.* In software development we usually find refinements between different abstraction levels. For instance, the first abstraction could refer to the concerns a system must deal with and the second one to the software artifacts which address such concerns (this could be extended to any phase in software development). As another example, the Model Driven Architecture (MDA) [22] provides a way to build software based on different refinements or transformations between models or artifacts belonging to different abstraction levels (e.g. Computational Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM)).
- *Elaboration.* We can find relationships between models of the same abstraction level. In such situations, we elaborate or add some extra information to a model in order to get a new model. For instance at requirements level we can elaborate a use case based on a previous one.
- *Representation.* In requirements engineering it is very common to have different representations of the same user needs. For instance, we can represent the requirements as statements extracted from a requirements elicitation document and we can also represent such requirements as viewpoints or use cases. We can link both kinds of representation by means of trace relationships.
- *Evolution.* With this type of dependencies we can relate gradual changes of software artifacts over time (as in adaptive maintenance). The <<evolves-to>> relationship exists between modified (structural and/or behavioral) elements in artifacts.

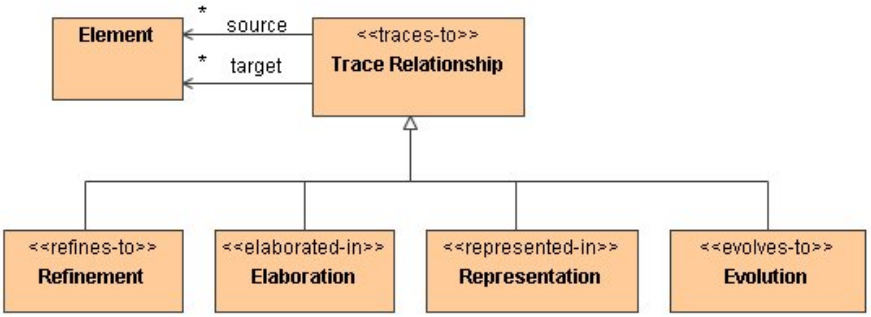

Figure 3. Traceability relationships model

In Table 1 we show some situations where crosscutting pattern can be applied. As we can see in this table, in the third column we show the different traceability types which can exist between source and target. These trace relations types belong to the simple model of traceability shown in Figure 3.

Table 1. Some examples of source and target domains

Examples	Source	Trace Relationship	Target
Ex. 1	Concerns	*are REFINED to*	Requirements Statements
Ex. 2	Concerns	*are REFINED to*	Use Cases
Ex. 3	Concerns	*are REFINED to*	Design Modules
Ex. 4	Use Cases	are *REFINED* to	Architectural Components
Ex. 5	Use Cases	*are REFINED to*	Design Modules
Ex. 6	PIM artifacts	*are REFINED TO*	PSM artifacts
Ex. 7	Requirements Statements	are *REPRESENTED* in	Viewpoints
Ex. 8	PIM artifacts	*are ELABORATED in*	PIM artifacts

2.1.2 Concepts based on Crosscutting Pattern

As we can see in Figure 2 and 3 there is a multivalued function from source elements to target elements. f': $S \longrightarrow T$ such that if $f'(s) = \{t\}$ then there exists a trace relation between s and t.

Analogously, we can define another multi-valued function g' that can be considered as the inverse of f'.

g': $T \longrightarrow S$ such that if $g'(t) = \{s\}$ then there exists a trace relation between s and t.

If f' is not a surjection, we can consider that T is the range of f', then g' is always a well-defined multi-valued function.

Obviously, f' and g' can be also represented as single-value functions considering that their codomains are the set of non-empty subsets of *Target* and *Source* respectively.

Let f: $Source \longrightarrow \mathscr{P}(Target)$ and g: $Target \longrightarrow \mathscr{P}(Source)$ be these functions defined by:

$\forall s \in Source, f(s) = \{t \in Target \ / \ there\ exists\ a\ trace\ relation\ between\ s\ and\ t\}$

$\forall t \in Target, g(t) = \{s \in Source \ / \ there\ exists\ a\ trace\ relation\ between\ s\ and\ t\}$

The concepts of scattering, tangling and crosscutting are defined as specific cases of these functions.

Scattering occurs when, in a mapping between source and target, a source element is related to multiple target elements.

Definition 1.[Scattering] *We say that an element s ϵ Source is* ***scattered*** *if card(f(s)) > 1.*

Tangling occurs when a target element is related to multiple source elements. In this case, we have focused on function *g*, i.e. the relation between target and source elements.

Definition 2.[Tangling] *We say that an element t ϵ Target is* ***tangled*** *if card(g(t))> 1.*

There is a specific combination of *scattering* and *tangling* which we call *crosscutting*. *Crosscutting* occurs when a source element is scattered over various target elements and at least one of these target elements is tangled.

Definition 3.[Crosscutting] Let *s1, s2 ϵ Source, s1 ≠ s2, we say that s1* ***crosscuts*** *s2 (s1 cc s2) if*

a) card(f(s1)) > 1

b) ∃ t ϵ f(s1): s2 ϵ g (t)

We do not require that the second source element (*s2*) is scattered. In that sense, our definition is not symmetric as definition in [21] (see Section 2.2).

In following paragraphs, we say that the definition 3 is the BCH-definition (Berg, Conejero and Hernández) of crosscutting.

From the previous definitions, we can follow a result that avoids the use of *g*. So, we work only with function *f*.

Lemma 1. *Let* s1, s2 *ϵ Source, s1 ≠ s2, then*

s1 crosscuts s2 if and only if card(f(s1)) > 1 and f(s1)∩f(s2) ≠∅.

Proof.

$$
\begin{aligned}
& s1 \; crosscuts \; s2 \\
& \Leftrightarrow card(f(s1)) > 1 \quad \wedge \quad \exists t \in f(s1) : s2 \in g(t) \\
& \Leftrightarrow card(f(s1)) > 1 \quad \wedge \quad \exists t \in f(s1) : t \in f(s2) \\
& \Leftrightarrow card(f(s1)) > 1 \quad \wedge \quad \exists t \in f(s1) \cap f(s2) \\
& \Leftrightarrow card(f(s1)) > 1 \quad \wedge \quad f(s1) \cap f(s2) \neq \varnothing
\end{aligned}
$$

□

2.2 Definition by Masuhara and Kiczales

In [21] the authors provide a very interesting model for defining how four different AOP mechanisms support modular implementation of crosscutting concerns. These mechanisms are based on a common framework which allows the authors to define what makes a technique aspect-oriented.

In addition, the authors also provide an interesting definition of crosscutting which can be compared with the concepts presented in previous section. The notion of crosscutting provided in [21] is focused on programming level, and it is based on two source languages A and B (one of them being aspect-oriented) and a target one called X (resulting of the weaving process of A and B). The authors take as input two different programs pA and pB written in A and B respectively. Then they define the term projection as follows: "*for a module mA (from pA), we say that the projections of mA into X is the set of join points identified by the A_{ID} elements within mA"*. A_{ID} refers to the means in A for identifying the join points in X (in object-oriented languages methods and field signatures). For more details see [21]. The authors use the canonical figures-display example [17] in the poincut-and-advice mechanisms to show these concepts in AspectJ (see Figure 4).

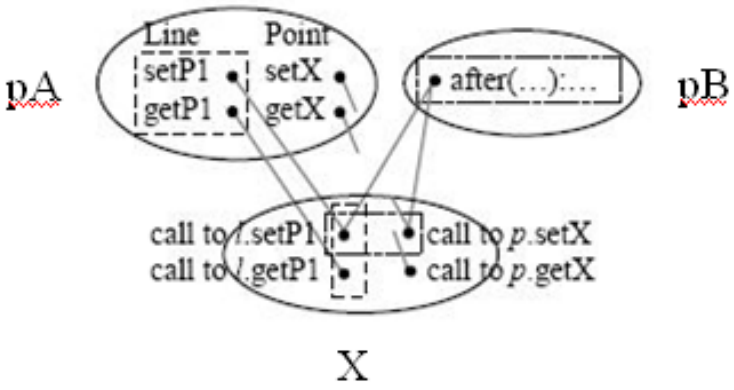

Figure 4. The `Point` class and the `display updating` advice crosscut each other in result domain X [23]

Then crosscutting is defined as follows: *For a pair of modules m_A and m_B we say that m_A crosscuts m_B with respect to X [the result domain] if and only if their projections onto X intersect, and neither of the projections is a subset of the other.* According to this definition crosscutting is a symmetric property.

We prove in following paragraphs that Masuhara and Kiczales definition of crosscutting (MK-definition) is a particular case of the definition 3 presented in previous subsection.

Let assume that

- *Source* = $\{m_A : m_A$ is a module of program $p_A\} \cup \{m_B : m_B$ is a module of program $p_B\}$
- *Target = {join points of X}*
- f: *Source* $\longrightarrow \mathcal{P}$ *(Target)* defined by *f(s)* is the projection of *s* onto *X*

This definition of *f* is independent of the fact that *s* will be a module of p_A or a module of p_B.

Thus, we can prove that any crosscutting situation detected by Masuhara and Kiczales in the context defined in [21] can be also detected with our definition.

Theorem 1. If there is a crosscutting situation using the MK-definition then there is also crosscutting using BCH-definition.

Proof. If there is a crosscutting situation using the MK-definition then there is a pair of modules m_A and m_B such that:

1. $f(m_A) \cap f(m_B) \neq \varnothing$
2. $f(m_A) \not\subset f(m_B)$
3. $f(m_B) \not\subset f(m_A)$

Obviously $card(f(m_A)) > 1$.

If $card(f(m_A) \leq 1$, as $f(m_A) \cap f(m_B) \neq \varnothing$ then $f(m_A) \subset f(m_B)$ and it is not true. Thus, $card(f(m_A) > 1$

Analogously, $card(f(m_B) > 1$.

Applying lemma 1, we have that m_A crosscut m_B according definition 3. □

Theorem 1 shows that MK-definition can be seen as a particular case of BCH-definition, being MK projections the mapping between Source and Target. Since MK-definition is focused on implementation level, we can say that definition based on Crosscutting Pattern is a generalization of it. This definition can be applied to any level or domain so that crosscutting can be identified in it.

Since our approach doesn't require that $f(mA) \not\subset f(mB) \wedge f(mB) \not\subset f(mA)$, this definition is less restrictive than other ones [1]. BCH definition only requires that the cardinality of the projection of mA onto X is larger than 1 (scattering), but the cardinality of mB onto X can be larger or equal than 1.The implications of this statement are important because of the set of crosscutting cases each definition could cover. That idea implies there would be some cases of crosscutting which definition based on Crosscutting Pattern identifies whereas other definitions do not. For example, certain tracing cases cannot be identified as crosscutting with MK-definition but with BCH, as we show below.

In [21], authors use the canonical figures-display example [17] to illustrate the application of their definition. This example can be also seen as a concrete application of the Observer Pattern defined in [14]. However, instead of considering the Display concern we may be interested in tracing the execution of all methods of Point or Line classes (Figure 5).

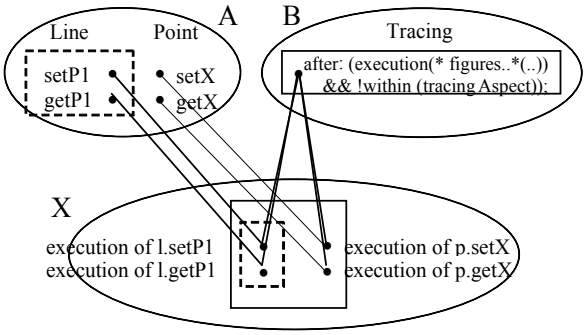

Figure 5. Projections of Line and Display advice according to Masuhara and Kiczales's definition

In that case, Masuhara and Kiczales's definition is applied as follows: projection of Line class onto X includes the execution of all methods of Line. The same is true for Point class. On the other hand, projections of advice include execution of all methods of Line and Point classes (in AspectJ execution join points are within the projection of the class that defines the method, as the authors explain in [21]). We can easily observe that projections of Line or Point are a subset of advice's one. Then, according to Masuhara and Kiczales's definition, subset condition is not accomplished in such an example and Line and Tracing do not crosscut each other. We could consider other monitoring techniques such as logging or profiling as similar examples [19]. However, as we do not require the subset condition, our definition identifies crosscutting in such a case. We just focus on cardinality of mA and intersection of both projections (mA and mB). This will be illustrated in next section by means of matrix representation of mappings between source and target.

To summarize the comparison between these definitions, we show the main differences:

- Since the definition based on the Crosscutting Pattern may be applied to any model or domain, it generalizes the definition presented in [21].
- The definition based on the Crosscutting Pattern is less restrictive than other one since it does not require the subset condition as in [21].
- The definition based on the Crosscutting Pattern does not consider crosscutting being a symmetric property whereas the definitions presented in [21] does.

The applicability of the definitions above depends on the goal of the crosscutting analysis.

[1] Note that $f(mA) \not\subset f(mB) \wedge f(mB) \not\subset f(mA)$ is equivalent to $f(mA) \setminus f(mB) \neq \varnothing \wedge f(mB) \setminus f(mA) \neq \varnothing$.

2.3 Definition by Lieberherr

In [34] Karl Lieberherr gives a definition of crosscutting: *"Two concerns crosscut if the methods related to those concerns intersect.(...) We say a method is related to a concern if the method contributes to the description, design, or implementation of the concern".* This definition can be considered as a particular case of MK-definition.

2.4 Definition by Ceccato et al.

In [28] Tonella and Ceccato use a mathematical tool to represent the relation between concerns and source code units; it is the formal concept analysis that will be introduce in following section.

2.4.1 ***Formal concept analysis.***

Formal concept analysis (FCA) is a branch of lattice theory that can be used to identify meaningful groupings of elements that have common properties. FCA takes as input a so-called **context**, which consists of a set of elements E, a set of properties P on those elements, and a Boolean incidence relation between E and P.

An example of such a context is given in Table 3, which relates different properties defined on integer numbers. Consider E = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, and P = {composite, even, odd, prime, square}.

Table 2. Dependency matrix that represents the relation between E and P.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Composite				X		X		X	X	X
Even		X		X		X		X		X
Odd	X		X		X		X		X	
Prime		X	X		X		X			
Square	X			X					X	

Starting from such a context, FCA determines maximal groups of elements and properties, called **concepts**, such that each element of the group shares the properties, every property of the group holds for all of its elements, no other element outside the group has those same properties, nor does any property outside the group hold for all elements in the group. Graphically, a concept corresponds to a maximal 'rectangle' containing only marks in the table, considering any permutation of the table's rows and columns.

A **concept lattice** can be built from a context. In the concept lattice, every node is a **concept** that is a pair containing both a property cluster and its corresponding object cluster.

The concept lattice shown in Figure 6 has been built from the context described in Table 2. Obviously, it is only a different way to represent the relation between E and P.

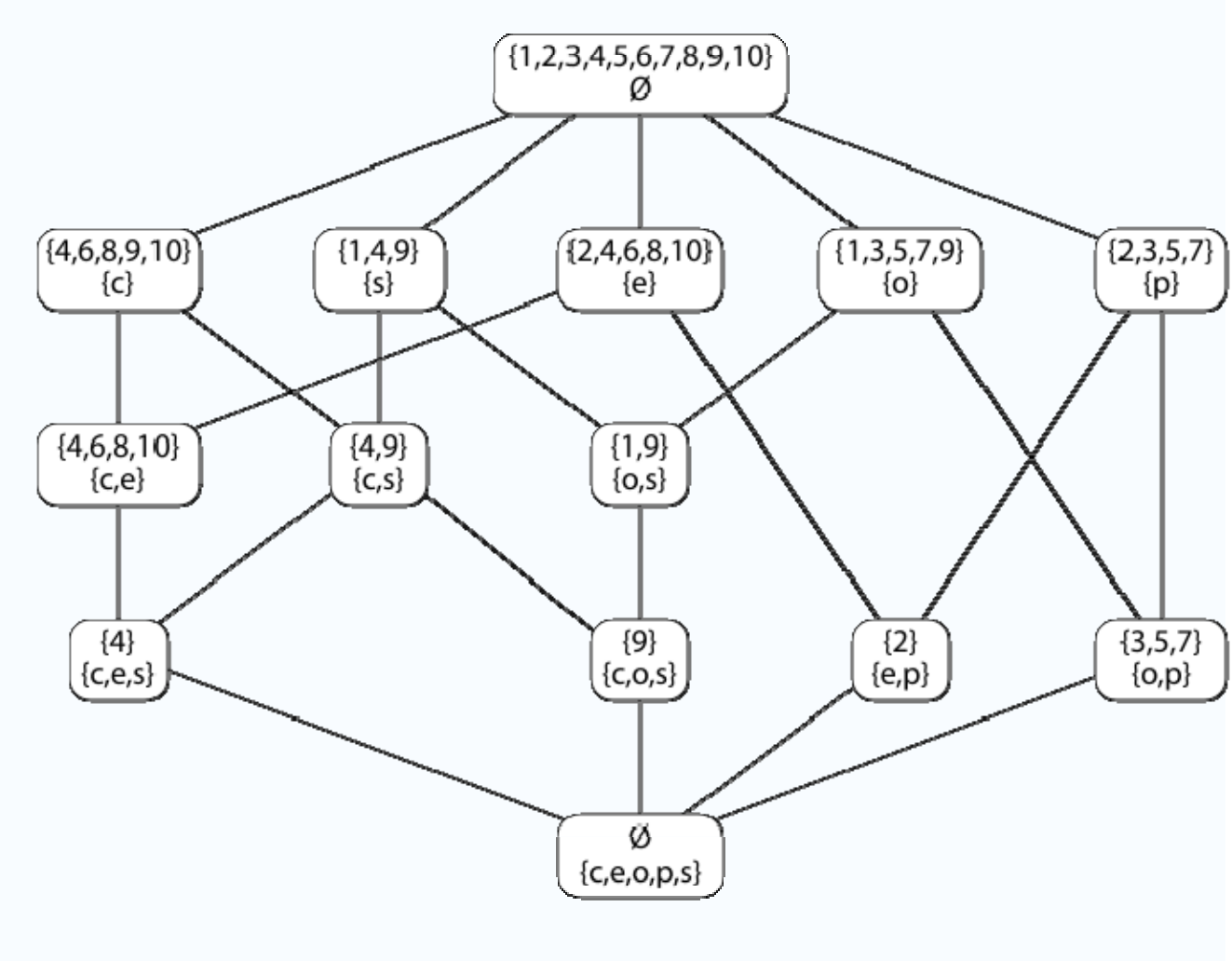

Figure 6. Concept lattice for the context describe in table 3

Formaly, a *concept* is defined to be a pair (E_i, P_i) such that

1. $E_i \in \mathscr{P}(E)$
2. $P_i \in \mathscr{P}(P)$
3. every element in E_i has every attribute in P_i
4. for every element in E that is not in E_i, there is a property in P_i that the element does not have
5. for every property in P that is not in P_i, there is an element in E_i that does not have that property

E_i is called the *extent* of the concept, and P_i is the *intent*.

Nodes in the concept lattice can be partially ordered by inclusion: if (E_i, P_i) and (E_j, P_j) are concepts, we define a partial order by saying that $(E_i, P_i) \leq (E_j, P_j)$ whenever $E_i \subseteq E_j$. Equivalently, $(E_i, P_i) \leq (E_j, P_j)$ whenever $P_j \subseteq P_i$. Every pair of concepts in this partial order has an unique greatest lower bound and an unique least upper bound. The greatest lower bound of (E_i, P_i) and (E_j, P_j) is the concept with elements $E_i \cap E_j$; it has as its properties $P_i \cup P_j$. The least upper bound of (E_i, P_i) and (E_j, P_j) is the concept with properties $P_i \cap P_j$; it has as its elements the set $E_i \cup E_j$.

2.4.2 Labels in the concept lattice

It is very interesting the idea of selecting elements and properties that label a given concept, they are those that characterize the concept most specifically.

More precisely, a concept **c** is labelled with an element **e** only if **c** is the most specific (i.e., lowest) concept having **e** in the extent. A concept **c** is labelled with a property **p** only if **c** is the most general (i.e., highest) concept having **p** in its intent.

We formally introduce this idea with the following functions:

$\alpha(c) = \{p \in P \mid c$ is the largest lower bound of the set of concepts that have **p** in its intent$\}$

$\beta(c) = \{e \in E \mid c$ is the least upper bound of the set of concepts that have **e** in its extent$\}$

Considering the previous example, the smallest concept including the number 3 is the one with objects {3, 5, 7}, and attributes {odd, prime}, then 3 is a label for this concept. The largest concept involving the attribute of being square is the one with objects {1,4,9} and attributes {square}, then square is a label for this concept. Thus:

$\alpha(\{3,5,7\}\{odd, prime\}) = \varnothing$ $\quad\quad$ $\beta(\{3,5,7\}\{odd, prime\}) = \{3,5,7\}$

$\alpha(\{1, 4, 9\}\{square\}) = \{square\}$ $\quad\quad$ $\beta(\{1,4,9\}\{square\}) = \{1,4,9\}$

2.4.3 Applying FCA to identify crosscutting situations

Formal concept analysis has been used to identify the computational units (i.e., procedures) that specifically implement a feature (i.e., requirement) of interest.

Execution traces obtained by running the program under given scenarios provided the input data. The executed methods are the elements of the concept analysis context, while execution traces associated with the use-cases are the properties[2]. In the resulting concept lattice, the use-case specific concepts are those labelled by at least one trace for some use-case (i.e. $\alpha(c)$ contains at least one element), while the concepts with zero or more properties as labels (those with an empty $\alpha(c)$) are regarded as generic concepts. Thus, use-case specific concepts are a subset of the generic ones.

Both use-case specific concepts and generic concepts carry information potentially useful for aspect mining, since they group specific methods that are always executed under the same scenarios.

2.4.4 Definition of crosscutting

A concern seed is a single source-code entity, such as a method, or a collection of such entities, that strongly connotes a crosscutting concern. A candidate seed is a potential concern seed.

Formally, a concept **c** is considered a candidate seed iff [8]:

- Scattering: $\exists$ m, m' $\in \beta(c) \mid pref(m) \neq pref(m')$
- Tangling: $\exists$ m $\in \beta(c), \exists$ m' $\in \beta(c') \mid c \neq c' \wedge pref(m) = pref(m')$

where pref(p) is the fully scoped name of the class containing the method p.

The first condition (scattering) requires that more than one class contributes to the functionality associated with the given concept. The second condition (tangling) requires that the same class addresses more than one concern.

[2] In [8], the authors claim that the executed methods are the properties of the concept analysis context. However, in the formal definition they define p and p' $\in \beta(c)$ (set of elements). We think this inconsistence is a typo. In order to clarify this issue, we considered here executed methods as the elements of the concept analysis context.

We prove in following paragraphs that Ceccato definition of crosscutting (C-definition) is a particular case of the definition 3 presented in subsection 2.1.2.

Let assume that

- *Source* is the set of concepts
- *Target* is the set of classes
- f: *Source* $\longrightarrow \mathcal{P}$ *(Target)* defined by $f(c) = \{pref(m) / m \in \beta(c)\}$

f(c) is the set of classes containing methods that labelled the concept **c**.

Thus, we can prove that any crosscutting situation detected by C-definition in the context defined in [8] can be also detected with our definition.

Theorem 2. If there is a crosscutting situation, the use of C-definition is equivalent to the use of BCH-definition.

Proof.

1. We prove that if there is a crosscutting situation using C-definition then there is also crosscutting using BCH-definition.

   The C-definition says that

   1. $\exists$ m, m' $\in \beta(c)$ / pref(m) $\neq$ pref(m')
   2. $\exists$ m $\in \beta(c)$, $\exists$ m' $\in \beta(c')$ / c $\neq$ c' $\wedge$ *pref(m)= pref(m')*

   Obviously, *card(f(c))* > 1, because pref(m), pref(m') $\in$ f(c) and, considering item 1 in C-definition, we have that pref(m) $\neq$ pref(m')

   Considering item 2, we have that $pref(m) \in f(c) \cap f(c') \Rightarrow f(c) \cap f(c') \neq \varnothing$.

   Applying lemma 1, we have that *c* crosscut *c'* according BCH-definition.

2. We prove that if there is a crosscutting situation using BCH-definition then there is also crosscutting using C-definition.

   Using lemma 1, the BCH-definition says that $\exists s1 \neq s2 \in$ *Source* such that

   a) $card(f(s1)) > 1$

   b) $f(s1) \cap f(s2) \neq \varnothing$.

   Considering item a, we have that f(s1) has at least two different elements.

   $cl1 \in f(s1) \quad \wedge \quad cl2 \in f(s1)$

   $\Rightarrow \exists m \in \beta(c) : cl1 = pref(m) \wedge \exists m' \in \beta(c) : cl2 = pref(m') \wedge m \neq m' \; (because \; cl1 \neq cl2)$

   Considering item b,

   $\exists \; cl \; \epsilon \; f(s1) \cap f(s2) \Rightarrow \exists m \in \beta(s1) \wedge m' \in \beta(s2) :$

   $cl = pref(m) = pref(m')$

   Then, s1 crosscut s2 according C-definition □

3. REPRESENTATION OF CROSSCUTTING

In this section, we describe how crosscutting can be represented by means of dependency graphs and an extension to traceability matrices. In the former we just represent the trace relationships between source

and target elements. In the latter trace relations are captured in a dependency matrix, representing the mapping between source and target. Since matrix representation allows the tool support by means of simple matrix operations, we focus on this representation. As an extension, we derive the crosscutting matrix from the dependency matrix. We describe how the crosscutting matrix can be constructed from the dependency matrix with some auxiliary matrices. This is illustrated with some examples.

3.1 Dependence graphs

Dependence graphs have been widely used in software engineering tasks such as program understanding, debugging, testing and maintenance. They have been mainly used at programming level showing control and data dependencies between code artefacts [11]. Even some approaches have emerged to adapt these graphs to represent aspect-oriented programs [29].

We may use such graphs to deal with the traceability links introduced in crosscutting pattern. As we showed in Section 2.1, a very intuitive and simple representation of mappings between source and target can be made by means of dependence graphs, so that crosscutting may be easily identified and represented by means of such graphs. We may distinguish several cases of mappings according to their cardinality between source and target:

- Injection: distinct source elements are related to distinct target elements (i.e. a *one-to-one*[3] function).
- Scattering: a source element is related to multiple target elements (i.e. a *one-to-many* function).
- Tangling: a target element is related to multiple source elements (i.e. a *many-to-one* function).
- Crosscutting: a target element is involved both in scattering and tangling (e.g. t3; scattering of s1 to t1, t3 and t4, and tangling of s1 and s3 in t3).

However, other representations of crosscutting may be possible. For instance, by means of traceability matrices, we can represent dependencies between source and target elements. In next section we show such matrices in order to identify and represent crosscutting. Matrix representation allows building automatic tools to find out crosscutting based on simple matrix operations.

3.2 Matrix representation

In terms of linear algebra, the relation between source elements and target elements can be represented in a special kind of traceability matrix [9] that we called dependency matrix. *A dependency matrix (source x target) represents the dependency relation between source elements and target elements (inter-level relationship)*. In the rows, we have the source elements, and in the columns, we have the target elements. In this matrix, a cell with 1 denotes that the source element (in the row) is *mapped* to the target element (in the column). Reciprocally this means that the target element *depends* on the source element. Scattering and tangling can easily be visualized in this matrix (see the examples below).

We define a new auxiliary concept *crosscutpoint* used in the context of dependency diagrams, to denote *a matrix cell involved in both tangling and scattering*. If there are one or more crosscutpoints then we say we have crosscutting.

Crosscutting between source elements for a given mapping to target elements, as shown in a dependency matrix, can be represented in a crosscutting matrix. *A crosscutting matrix (source x source) represents the crosscutting relation between source elements, for a given source to target mapping (represented in a dependency matrix)*. In the crosscutting matrix, a cell with 1 denotes that the source element in the row is crosscutting the source element in the column. In section 3.3 we explain how this crosscutting matrix can be derived from the dependency matrix.

---

[3] This name is best avoided, since some authors understand it to mean a bijective function

A crosscutting matrix should not be confused with a coupling matrix. A *coupling matrix* shows coupling relations between elements at the same level or abstraction (intra-level dependencies). In some sense, the coupling matrix is related to the design structure matrix [2]. On the other hand, a crosscutting matrix shows crosscutting relations between elements at one level with respect to a mapping onto elements at some other level (inter-level dependencies).

We now give an example and use the dependency matrix and crosscutting matrix to visualize the definitions (S denotes a scattered source element - a grey row; NS denotes a non-scattered source element; T denotes a tangled target element - a grey column; NT denotes a non-tangled target element). The example is shown in Table 3.

Table 3. Example dependency and crosscutting matrix with tangling, scattering and one crosscutting

dependency matrix

		target				
		t[1]	t[2]	t[3]	t[4]	
source	s[1]	1	0	1	1	S
	s[2]	0	1	0	0	NS
	s[3]	0	0	1	0	NS
		NT	NT	T	NT	

crosscutting matrix

		source		
		s[1]	s[2]	s[3]
source	s[1]	0	0	1
	s[2]	0	0	0
	s[3]	0	0	0

In this example, we have one scattered source element s[1] and one tangled target element t[3]. We apply our definition of crosscutting and arrive to the crosscutting matrix. Source element s[1] is crosscutting s[3] (because s[1] is scattered over [t[1], t[3], t[4]] and s[3] is in the tangled one of these elements, namely t[3]). The reverse is not true: the crosscutting relation is not symmetric. The example is depicted in the diagrams (Table 3).

3.3 Constructing crosscutting matrices

In this section, we describe how to derive the crosscutting matrix from the dependency matrix. We use a more extended example than the previous ones. We now show an example with more than one crosscutpoints, in this example 8 points (see Table 4; the dark grey cells).

The crosscutting matrix shows that the crosscutting relation is not symmetric. For example, s[1] is crosscutting s[3], but s[3] is not crosscutting s[1] because s[3] is not scattered (scattering is a necessary condition for crosscutting).

Table 4. Example dependency matrix with tangling, scattering and several crosscuttings

dependency matrix

		target						
		t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	
source	s[1]	1	0	0	1	0	0	S
	s[2]	1	0	1	0	1	1	S
	s[3]	1	0	0	0	0	0	NS
	s[4]	0	1	1	0	0	0	S
	s[5]	0	0	0	1	1	0	S
		T	NT	T	T	T	NT	

crosscutting matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	0	1	1	0	1
	s[2]	1	0	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	0	0
	s[5]	1	1	0	0	0

Based on the dependency matrix, we define some auxiliary matrices: the *scattering matrix* (source x target), and the *tangling matrix* (target x source). These two matrices are defined as follows (for our example in Table 4, these matrices are shown in Table 5):

- In the scattering matrix a row contains only dependency relations from source to target elements if the source element in this row is scattered (mapped onto multiple target elements); otherwise the row contains just zero's (no scattering).

- In the tangling matrix a row contains only dependency relations from target to source elements if the target element in this row is tangled (mapped onto multiple source elements); otherwise the row contains just zero's (no tangling).

Table 5. Scattering and tangling matrices for dependency matrix in Table 4

scattering matrix

		target					
		t[1]	t[2]	t[3]	t[4]	t[5]	t[6]
source	s[1]	1	0	0	1	0	0
	s[2]	1	0	1	0	1	1
	s[3]	0	0	0	0	0	0
	s[4]	0	1	1	0	0	0
	s[5]	0	0	0	1	1	0

tangling matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
target	t[1]	1	1	1	0	0
	t[2]	0	0	0	0	0
	t[3]	0	1	0	1	0
	t[4]	1	0	0	0	1
	t[5]	0	1	0	0	1
	t[6]	0	0	0	0	0

We now define the crosscutting product matrix, showing the frequency of crosscutting relations. *A crosscutting product matrix (source x source) represents the frequency of crosscutting relations between source elements, for a given source to target mapping.* The crosscutting product matrix is not necessarily symmetric. The *crosscutting product matrix* ccpm can be obtained through the matrix multiplication of the scattering matrix sm and the tangling matrix tm: ccpm = sm . tm where ccpm [i][k] = sm[i][j] tm[j][k].

In this crosscutting product matrix, the cells denote the frequency of crosscutting. This can be used for quantification of crosscutting (crosscutting metrics). The frequency of crosscutting in this matrix should be seen as an upper bound. In actual situations, the frequency can be less than the frequency from this matrix analysis, because in the matrix we abstract from scattering and tangling specifics. In the crosscutting matrix, a matrix cell denotes the occurrence of crosscutting; it abstracts from the frequency of crosscutting.

The *crosscutting matrix* ccm can be derived from the crosscutting product matrix ccpm using a simple conversion: ccm[i][k] = if (ccpm[i][k] > 0) /\ (i ≠ j) then 1 else 0.

The crosscutting product matrix and the crosscutting matrix for the example are given in Table 6. In this example, there are no cells in the crosscutting product matrix larger than 1, except on the diagonal where it denotes a crosscutting relation with itself, which we disregard here. In the crosscutting matrix, we put the diagonal cells to 0. Obviously, this is because we interpret a source element can't crosscut itself.

Table 6. Crosscutting product matrix and crosscutting matrix for dependency matrix in Table 4

crosscutting product matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	2	1	1	0	1
	s[2]	1	3	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	1	0
	s[5]	1	1	0	0	2

crosscutting matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	0	1	1	0	1
	s[2]	1	0	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	0	0
	s[5]	1	1	0	0	0

As we can see in crosscutting matrix in Table 6, there are now 10 crosscutting relations between the source elements. The crosscutting matrix shows again that the crosscutting relation is not symmetric. For example, s[1] is crosscutting s[3], but s[3] is not crosscutting s[1] because s[3] is not scattered (scattering and tangling are necessary but not sufficient condition for crosscutting).

For convenience, these formulas can be calculated automatically by means of simple mathematic tools (such as Excel). By filling in the cells of the dependency matrix, the other matrices are calculated automatically.

3.4 Case Analysis of Crosscutting

Once we have defined scattering, tangling and crosscutting, we may discuss now a case analysis of possible combinations according to our definition. Assuming that the properties tangling, scattering, and crosscutting may be true or false, there are 8 combinations (see Table 7). Each case addresses a certain mapping from source to target. However, crosscutting requires tangling and scattering, which eliminates 3 of these combinations (Cases 6, 7 and 8: not feasible). There are five feasible cases listed in the table. In Case 4, we have scattering and tangling in which no common elements are involved. With our definition of crosscutting, we disentangle the cases with just tangling, just scattering and on the other hand crosscutting. Our proposition is that tangling and scattering are necessary but not sufficient conditions for crosscutting.

Table 7. Feasibility of combinations of tangling, scattering and crosscutting

	tangling	scattering	crosscutting	feasibility
Case 1	No	no	no	feasible
Case 2	Yes	no	no	feasible
Case 3	No	yes	no	feasible
Case 4	Yes	yes	no	feasible
Case 5	Yes	yes	yes	feasible
Case 6	No	no	yes	not feasible
Case 7	No	yes	yes	not feasible
Case 8	Yes	no	yes	not feasible

In order to illustrate the different possibilities, we discuss now how to apply the framework to some simple examples. The first example is extracted from [28], where the authors use the definition presented in Section 2.4.4 to identify crosscutting concerns at programming level. The example application consists of several classes that implement a simple Binary Search Tree. The main functionalities of the application are the *insertion* of elements in the data structure and the *search* of a particular element. The class diagram is shown in Figure 7.

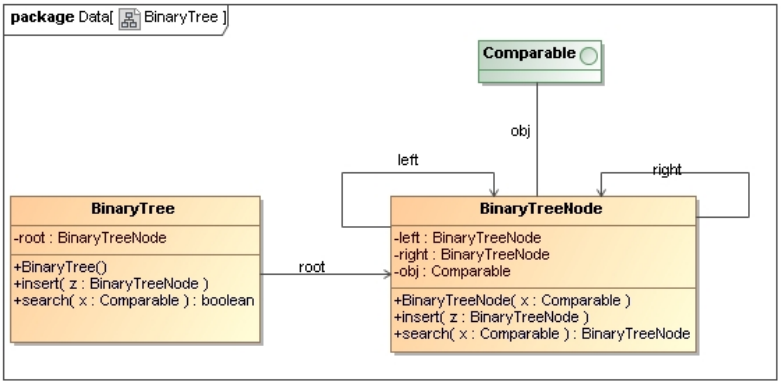

Figure 7. Binary Search Tree class diagram

In [28], the authors present a table where the two main concerns of the system, *insertion* and *search*, are related to the methods that contribute to such functionalities. Assuming that the *search* is performed in a pre-loaded binary tree, these methods are presented in Table 8.

Table 8. Relation between the main concerns and the executed methods for these concerns

	Insertion
m1	BinaryTree.BinaryTree()
m2	BinaryTree.Insert(BinaryTreeNode)
m3	BinaryTreeNode.insert(BinaryTreeNode)
m4	BineryTreeNode.BinaryTreeNode(Comparable)
	Search
m1	BinaryTree.BinaryTree()
m5	BinaryTree.search(Comparable)
m6	BinaryTreeNode.search(Comparable)

Having the concerns and the methods that contribute to them as source and target domains respectively, we build the dependency matrix shown in Table 9. We have selected methods as the granularity level for the target elements. As we can see in this matrix, the BinaryTree.BinaryTree() method is executed for both the *insertion* and the *search* concerns. The existence of this method implies that our framework identifies both concerns as crosscutting (see crosscutting matrix in Table 10).

Table 9. Dependency matrix for the BST application

		methods					
		m1	m2	m3	m4	m5	m6
concerns	insertion	1	1	1	1	0	0
	search	1	0	0	0	1	1

Table 10. Crosscutting matrix for the BST application

		concerns	
		insertion	search
concerns	insertion	0	1
	search	1	0

The example explained above belongs to the fifth category of the eight possible combinations presented in Table 7 (i.e. scattering, tangling and crosscutting). However, we may find different situations with just scattering or just tangling and not crosscutting. For instance, since in [28] the authors consider the *search* concern having a pre-loaded tree, we do not consider that the constructor of BinaryTree class contributes to such a functionality. In that case, we remove the mapping from *search* concern to method m1. The new dependency and crosscutting matrices are shown in Table 11 and Table 12 respectively.

Table 11. New dependency matrix for the BST

		methods					
		m1	m2	m3	m4	m5	m6
concerns	insertion	1	1	1	1	0	0
	search	0	0	0	0	1	1

Table 12. New crosscutting matrix for the BST

		concerns	
		insertion	search
concerns	insertion	0	0
	search	0	0

As we can see in the dependency matrix of Table 11, we may have source elements scattered over different target elements without having crosscutting. Although usually we encounter scattering and tangling together, the utilization of a formal definition allows the differentiation of these concepts identifying such exceptional situations (with only one of the needed conditions to have crosscutting). This last situation belongs to the third case or category of Table 7.

Even if we consider that the constructor of the BinaryTree class contributes to the searching functionality, we could find a case where a source element is scattered over different target elements and there is not crosscutting. For instance, consider the same BST system explained above without the searching functionality. In that case, the *insertion* concern would be scattered over some methods and classes, we do not consider such a concern as being crosscutting. Obviously, if there is just one concern,

it could not crosscut to any other concern. However, note that our formal definition of crosscutting works properly in that case (that is what we are proofing).

In order to show a different case with tangling and not crosscutting, we show now a new simple example, a calculator with remote access. We apply the framework at concern level with respect to the design level (represented in a UML class diagram). The case study consists of a distributed Java application which allows a user to calculate the sum of integer numbers. The distribution is accomplished by means of sockets. The MVC pattern [6] is applied in order to perform a separation of representation and control concerns from the functional concerns of an application. In order to study the crosscutting in this case, we consider three main concerns in the system: *Client side distribution*, *Server side distribution* and *Calculation*. We take these concerns as source elements in our dependency matrix and the UML design classes are considered to be the target elements.

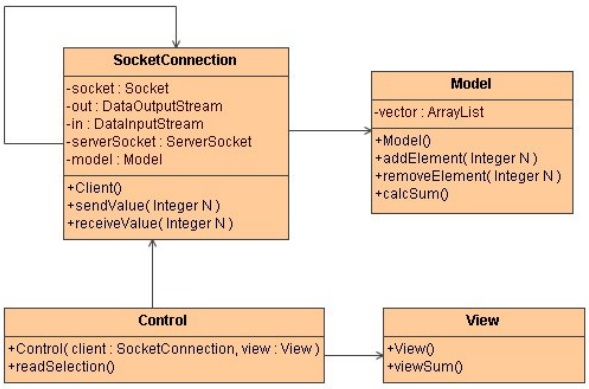

Figure 8. UML class diagram of Remote Calculator

In Figure 8 we show a UML class diagram representing the design. We have developed the main functionality regarding the socket concerns in a class called SocketConnection. This class just performs the remote connection and sends and receives integer values. We may say that this class has a low cohesion. Depending on the operation (sending or receiving), this class will invoke methods of the other classes. The Model, View and Control classes perform the actions to sum the integer, read user's selections and shows the results on screen respectively. Therefore, the application has a good separation between model (a class with a vector of numbers and which performs the sum), view (a class which shows the result on the screen) and control (a class which reads the user's inputs). Although such classes are coupled by means of method calls, their level of cohesion is high because each class is only addressing its main functionality (concern).

So, taking such a decomposition (in classes) and applying the framework, we obtain the dependency matrix shown in Table 13. As we can see in the matrix, concerns *Client side distribution* and *Server side distribution* are tangled in the same class SocketConnection, whereas *Calculation* concern is scattered over the other classes. However, as can be seen in the table, the matrix has no crosscutpoints. By means of the operations described in Section 3.3 we obtain the crosscutting matrix shown in Table 13: there are no crosscutting concerns in the system.

In many situations, we have tangling, scattering and at the same time crosscutting. With our definitions, we clearly distinguished scattering and tangling from crosscutting and, as we stated in Section 2.1, scattering and tangling are necessary but not sufficient conditions for crosscutting. The analysis depends on the chosen decomposition of source and target, other decompositions being feasible.

Table 13. Dependency and crosscutting matrix for the Remote Calculator

dependency matrix

	classes				
concerns	SocketConnection	Model	View	Control	
Distribution-to-Client	1	0	0	0	NS
Distribution-to-Server	1	0	0	0	NS
Calculation	0	1	1	1	S
	T	NT	NT	NT	

crosscutting matrix WRT[4] classes

	Concerns		
concerns	Distribution-to-Client	Distribution-to-Server	Calculation
Distribution-to-Client	0	0	0
Distribution-to-Server	0	0	0
Calculation	0	0	0

4. CROSSCUTTING AND TRANSITIVITY OF DEPENDENCIES

In this section we consider the transitivity of dependencies between levels and within the same level respectively. Such dependencies are based on different transitive relations that can be observed between source and target elements.

4.1 Transitivity of inter-level dependencies

Usually we encounter a number of consecutive levels or phases in software development. In MDA [21], we have transformations from Platform Independent Models, Platform Specific Models to Implementation Models. From the perspective of software life cycle phases, we could distinguish Domain Analysis, Concern Modelling, Requirement Analysis, Architectural Design, Detailed Design, and Implementation.

We consider here the cascading of two crosscutting patterns: the target of the first pattern serves as source for the second one. For convenience, we call the first target our intermediate level, and our second target just target (see Figure 9).

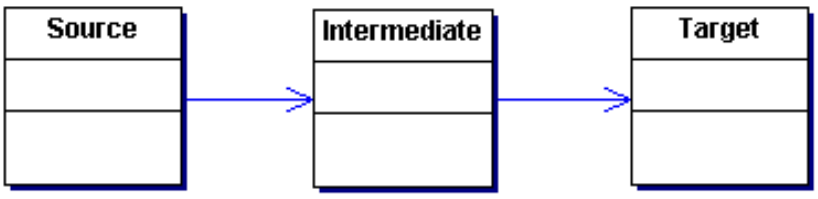

Figure 9. Two Cascaded Crosscutting Patterns

Each of these refinements can be described with a dependency matrix. We describe how to combine two consecutive dependency matrices, in an operation we call cascading. Cascading is an operation on two dependency matrices resulting in a new dependency matrix, which represents the dependency relation between source elements of the first matrix and target elements of the second matrix.

For cascading, it is essential to define the transitivity of dependency relations. Transitivity is defined as follows. Assume we have a source, an intermediate level, and a target. There is a dependency relation

[4] WRT are the abbreviation of "with respect to"

between an element in the source and an element in the target if there is some element at the intermediate level that has a dependence relation with this source element and a dependency relation with this target element. In other words, the transitivity dependency relation R for source s, intermediate level u and target t, and card(u) is the number of elements in u:

$$\exists\, k \in (1..card(u)) : (s[i]\ R\ u[k]) \wedge (u[k]\ R\ t[m]) \Rightarrow (\ s[i]\ R\ t[m]\ )$$

We can also formalize this relation in terms of the dependency matrices. Assume we have three dependency matrices m1 :: s × u and m2 :: u × t and m3 :: s × t, where s is the source, u is some intermediate level, card(u) is the cardinality of u, and t is the target. The cascaded dependency matrix m3 = m1 cascade m2

Then, *transitivity* of the dependency relation is defined as follows:

$$\exists\, j \in (1..card(u)): m1[i,j] \wedge m2[j,k] \Rightarrow m3[i,k]$$

In terms of linear algebra, the dependency matrix is a relationship between two given domains, source and target (see section 2.1.1). Accordingly, the cascading operation can be generalized as a composition of relationships as follows. Let Dom_K, $k = 1..n$, be n domains, and let f_i be the relationship between domains Dom_i and Dom_{i+1}, $1 \le i < n$, denoted as $Dom_i \xrightarrow{f_i} Dom_{i+1}$. Let Source and Target be the domains Dom_1 and Dom_n, respectively. Consequently, we have the following relationship between the domains: $Source \xrightarrow{f_1} Dom_2 \xrightarrow{f_2} Dom_3 \xrightarrow{f_3} \ldots Dom_{n-1} \xrightarrow{f_{n-1}} Target$

As a result, the dependency relationship between the Source and the Target is defined as $DM \equiv f_{n-1} \circ f_{n-2} \circ \ldots \circ f_1$. In this way, the dependence matrix between a source and target is obtained through matrix multiplication of the dependency matrices that represents each f_i, $1 \le i < n$.

Table 14. Two dependency matrices that will be cascaded

dependency matrix 1

	requirement			
concern	r[1]	r[2]	r[3]	r[4]
c[1]	1	0	0	1
c[2]	0	1	0	0
c[3]	0	0	1	1

dependency matrix 2

	module				
requirement	m[1]	m[2]	m[3]	m[4]	m[5]
r[1]	1	0	0	0	1
r[2]	0	1	0	0	0
r[3]	0	1	1	0	0
r[4]	0	0	0	1	1

As an example, we explain the cascading two dependency matrices: one for concerns x requirements and one for requirements x modules. The two dependency matrices are shown in Table 14. The first dependency matrix relates concerns with requirements. The second dependency matrix relates requirements with modules. The resulting dependency matrix relates concerns with modules (see Table 15). This matrix can be used to derive the crosscutting matrix for concern × concern with respect to modules. The crosscutting matrix in Table 15 is not symmetric. Based on this matrix we conclude, for the given dependency relations between concerns and modules, that: concern c[1] is crosscutting concern c[3]; concern c[2] does not crosscut any other concern; concern c[3] is crosscutting concerns c[1] and c[2].

Table 15. The resulting dependency matrix and crosscutting matrix based on cascading of the matrices in Table 14

resulting dependency matrix

	module				
concern	m[1]	m[2]	m[3]	m[4]	m[5]
c[1]	1	0	0	1	2
c[2]	0	1	0	0	0
c[3]	0	1	1	1	1

crosscutting matrix

	concern		
concern	c[1]	c[2]	c[3]
c[1]	0	0	1
c[2]	0	0	0
c[3]	1	1	0

From this description, it is clear that cascading can be used for *traceability* analysis across multiple levels, e.g. from concerns to implementation elements, via requirements, architecture and design (c.f. [26]).

4.2 Transitivity of intra-level dependencies

Elements at a certain level usually have some relationship with other elements at the same level (intra-level relationships): they are coupled. There are many coupling types: generalisation/specialisation, aggregation, data coupling, control coupling, message coupling, and so on. In case of a dependency relation of a source element and a target element, which itself is coupled to a second target element, one could conceive also a dependency relation between the source element and the second target element.

Intra-level trace dependencies combined with inter-level trace dependencies may cause dependencies, which we call an *indirect trace dependency* based on a pseudo-transitivity. Assume source element s[i] has a coupling relation R' with source element s[j]. Moreover source element s[j] has a dependency relation R with target element t[k]. Then the indirect dependency relation is $(s[i] \; R' \; s[j]) \wedge (s[j] \; R \; t[k]) \Rightarrow (s[i] \; R' \circ R \; t[k])$. In a similar way, assume source element s[i] has a dependency relation R with target element t[j] and target element t[j] is coupled with target element t[k] by means of R'. In that case the indirect dependency relation is $(s[i] \; R \; t[j]) \wedge (t[j] \; R' \; t[k]) \Rightarrow (s[i] \; R \circ R' \; t[k])$.

One should clearly distinguish the direct (inter-level) dependency relation from this indirect dependency relation. Our framework is focused on direct trace relationships.

5. Aspect-oriented metrics

We describe in this section a new topic where the framework provides important benefits: aspect-oriented metrics. There are several works that have explained the need for adapting the traditional object oriented metrics to the new aspect oriented paradigm. For instance, we need metrics to measure the degree of crosscutting in a system. In [25] the authors propose a framework where they define several metrics in terms of, on one hand, separation of concerns and, on the other hand, cohesion, coupling and size. In [10] some similar metrics suite is defined in order to measure the degree of scattering and tangling in some components. In this last work, the authors take as base for the metrics the definition of crosscutting presented in our previous work [4]. In following sections, we explain in more detail these metrics and show how they may be represented, visualized and extended by means of our framework.

5.1 Metrics by García et al.

In [25], the authors presented an aspect oriented metrics suite. This suite is based on the previous work of the authors presented in [15]. While the latter is focused on the assessment modularity at programming level, the former is focused on the architecture level. In the work presented in [25], the

metrics suite is based on the concept of architectural concern. The authors claim that the framework presented relies on evaluating the modularization of architectural concerns quantifying separation of concerns and their interactions. As an example, they establish a metric to assess the diffusion of a concern over the architectural artifacts. In Table 16 we show the whole metrics suite that they set. This table is extracted from [25].

Table 16. Metrics suite defined in [25]

Attribute	Metric	Definition
Concern Diffusion	Concern Diffusion over Architectural Components (CDAC)	It counts the number of architectural components which contributes to the realization of a certain concern.
	Concern Diffusion over Architectural Interfaces (CDAI)	It counts the number of interfaces which contributes to the realization of a certain concern.
	Concern Diffusion over Architectural Interfaces (CDAI)	It counts the number of operations which contributes to the realization of a certain concern.
Coupling Between Architectural Concerns	Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which the assessed concerns share at least a component.
	Interface-level Interlacing Between Concerns (IIBC)	It counts the number of other concerns with which the assessed concerns share at least an interface.
	Interface-level Interlacing Between Concerns (IIBC)	It counts the number of other concerns with which the assessed concerns share at least an operation.
Coupling Between Components	Afferent Coupling Between Components (AC)	It counts the number of components which require service from the assessed Coupling component
	Efferent Coupling Between Components (EC)	It counts the number of components from which the assessed component requires service.
Component Cohesion	Lack of Concern-based Cohesion (LCC)	It counts the number of concerns addressed by the assessed component.
Interface Complexity	Number of Interfaces	It counts the number of interfaces of each component.
	Number of Operations	It counts the number of operations in the interfaces of each component.

As we can see in the previous table the metrics are classified into five different categories. The first category, *Concern Diffusion* is focused on the relation between concerns and the architectural elements. In terms of our framework, we can say that this metric is related to the trace relation between source and target. In particular it assesses the cardinality of the relation from source to target elements. The second and third categories are focused on the relation between elements of the same domain. Thus, they are based on intra-level relations. While the *Coupling Between Architectural Concerns* metric is based on the relations between elements of the source domain (concerns), the *Coupling Between Components* is based on relations between elements of the target domain (architectural artifacts). The fourth category, *Component Cohesion* is similar to the first one. It is focused on the relation between source and target elements. However, in this case, the metrics assesses the cardinality of the relation from target to source elements. Finally, the purpose of the *Interface Complexity* category is to measure the number of interfaces and operations of the architectural components. Therefore this metric is just a way to measure the size of the decomposition selected on the target domain.

5.2 Metrics by Aho et al.

In [10], the authors establish a concern model to define crosscutting and set a suite of aspect oriented metrics to complement the traditional metrics (coupling and cohesion). The concern model defined in [10] is based on our previous work [4]. In particular they used the relations between source and target

domains explained in Section 2.1.1 to define crosscutting. Unlike our definition, they don't consider tangling as a necessary condition to have crosscutting. They define crosscutting as follows: *a crosscutting concern is a scattered concern, i.e., a concern related to multiple target elements.* The authors define two main metrics: *Degree of Scattering (DOS)* and *Degree of Focus (DOF)*. Other auxiliary metrics are used to obtain the DOS and DOF ones.

Firstly, the authors define *Concentration* (CON) as *a measure of the number of statements related to a concern within a specific component*. Thus, CON focuses on a particular source (s) and target element (t) and assesses how such a target element contributes to the realization of the source element:

$$CONC(s,t) = \frac{SLOCs\ in\ component\ t\ related\ to\ concern\ s}{SLOCs\ related\ to\ concern\ s}$$

The authors of this metric claim that CON does not give a sense for how scattered a concern is and different concerns may not be compared. Then, they introduce a new metric called Degree of Scattering (DOS). DOS is a measure *of the variance of the concentration of a concern over all components of the system*. A value of DOS close to 1 indicates that this concern (s) has a high degree of scattering while a value close to 0 indicates that this concern is not scattered and is well encapsulated into an entity:

$$DOS(s) = 1 - \frac{|T| \sum_t^T \left(CONC(s,t) - \frac{1}{|T|} \right)^2}{|T| - 1}$$

where $|T|$ is the number of components. The authors in [10] also use the average of the DOS (ADOS) metric (averaging DOS over all the concerns) to measure modularity of the system.

The metrics mentioned above are focused on the relation from source to target elements. The authors also provide some metrics focused on the relation from target to source elements. In particular they define the *Dedication* metric which *assesses the number of statements of a component (t) related to a particular concern (s)*:

$$DEDI(t,s) = \frac{SLOCs\ in\ component\ t\ related\ to\ concern\ s}{SLOCs\ related\ to\ component\ t}$$

Again the authors provide a different metric to measure *how well the concerns are separated in a component.* This metric is called the *Degree of Focus* (DOF) and is *a variance of the dedication of a component to every concern with respect to the worse case:*

$$DOF(t) = \frac{|S| \sum_s^S \left(DEDI(t,s) - \frac{1}{|S|} \right)^2}{|S| - 1}$$

where $|S|$ is the number of components. In this case, a value of DOF close to 0 indicates that the component's attention is uniformly divided among every concern while a value close to 1 indicates that the component is focused just on one concern. The authors also use the average of DOF (ADOF) to provide an overall indication of the separation of concerns in the program.

5.3 Relation between crosscutting pattern and metrics

As we have shown in the previous sections, the metrics defined in [25] and [10] are closely related to the relation between source and target domains represented in the crosscutting pattern. In some cases the metrics are focused on the relation from source to target while in other cases they are related to the

inverse relation. As we said early on, the metrics defined in those works may be visualized and integrated into the framework presented in this report. In particular, we represent some of the metrics presented by García et al. and by Aho et al. in our matrices. We use the scattering and tangling matrices to represent the metrics that are directly related to scattering and tangling properties respectively.

5.3.1 Metrics for Scattering

Firstly we use the scattering matrix to represent the *Concern Diffusion* and the *Degree of Scattering* metrics presented in [25] and [10] respectively. The *Concern Diffusion* metric is divided into three different metrics depending on the granularity of the architectural artefacts selected: components, interfaces or operations level. The rows in the scattering matrix presented in Section 3.3 provide similar information about the diffusion of a concern over the target elements. For instance, in Table 17 we show a part of the scattering matrix for a concurrent file versioning system (CFVS) [5]. We have applied our framework in such a system in order to identify crosscutting concerns. For instance, we can see how the `Synchronization` or the `Concurrency` concerns (among others) are scattered over different use cases. Again, depending on the granularity of the target elements selected we may use a different metric of the *Concern Diffusion* category.

Table 17. Part of the scattering matrix for the CFVS

		Use cases																
		Insert File	Retrieve File	Change File	Commit File	Update File	Conflict Management	Show Message	Show Differences	Remove File	Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message
Concerns	Insert File	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Branch a Set of Files	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Merge Set of Files	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Persistence	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
	Data Representation	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1
	Synchronization	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
	Logging	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Security	1	1	0	1	1	0	1	1	1	1	1	1	1	1	0	1	1
	Concurrency	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

In order to assess how much scattered a particular source element is, we have also defined a metric based on the scattering matrix. We called this metric just Scattering (SCAT). This metric is calculated as follows:

$$SCAT(s_i) = \frac{\left(\sum_{j=1}^{|T|} scat_{ij}\right) - 1}{|T|}$$

where s_i is the source element of row i in the scattering matrix, $|T|$ is the number of target elements in the system, and $scat_{ij}$ represents the value of the cell ij of the scattering matrix. SCAT may have values between zero and one. When a source element (i.e. a concern) presents a value of zero in this metric this element is well encapsulated and is not scattered over the system while a value close to one indicates that the source element is highly scattered over the target ones. In order to have a global metric for how much scattering the system has we define the concept of Global Scattering (GSCAT) which is obtained just calculating the average of the SCAT values for each source elements:

$$GSCAT = \frac{\sum_{i=1}^{|S|} SCAT(s_i)}{|S|}$$

where $|S|$ is the number of source elements in the system.

Secondly, the *Degree of Scattering* metric (defined in [10]) may be also calculated taking as input the values represented in the scattering matrix. As we explained in Section 5.2 the *Degree of Scattering* metric is based on an auxiliary metric called *Concentration*. *Concentration* is defined as the relation between lines of code of a component contributing to a particular concern and the total number of lines of code of the component. This metric may be easily represented in our scattering matrix just adapting the matrix to the metric. Since both scattering and tangling are binary matrices, we represent cells with just 1 or 0. However we may change these cells to represent the relation defined by *Concentration*. For instance, a cell in the scattering matrix with 0.5 represents that the component of this column is performing the half of lines of code belonging to the concern of the corresponding row. In Table 18 we show an example of several rows containing different values of the *Concentration* metric for some particular concerns. Obviously, having these values, the calculation of *Degree of Scattering* is very simple.

Table 18. Scattering matrix with Concentration values

		Use cases																
		Insert File	Retrieve File	Change File	Commit File	Update File	Conflict Management	Show Message	Show Differences	Remove File	Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message
	Security	0,05	0,05	0	0,05	0,05	0	0,05	0,05	0,05	0,05	0,05	0,05	0,05	0,2	0	0,2	0.05
	Concurrency	0	0	0	0,15	0,15	0,7	0	0	0	0	0	0	0	0	0	0	0

5.3.2 Metrics for Tangling

Similarly to the scattering metrics mentioned above, the metrics related to tangling defined in [25] and [10] may be also represented by means of the tangling matrix presented in Section 3.3. On one hand, the *Component Cohesion* metric presented in [25] is represented by a column of the tangling matrix. In a column of the matrix mentioned we may see the different concerns that a particular component is addressing. In Table 19 we show part of the tangling matrix for the CFVS.

As we did for the scattering in the system, we have defined a metric for assessing the tangling in the system. This metric is based on the values of the tangling matrix. We called this metric Tangling (TANG). This metric allows a developer to assess the number of source elements being addressed by a particular target element. It is defined as:

$$TANG(t_j) = \frac{\left(\sum_{i=1}^{|S|} tang_{ij}\right) - 1}{|S|}$$

where t_j is the target element of column j in the tangling matrix, $|S|$ is the number of source elements in the system, and $tand_{ij}$ represents the value of the cell ij of the tangling matrix. Again, TANG may have values between zero and one where zero indicates lack of tangling and a value close to 1 indicates a high degree of tangling. Similarly to the Global Scattering metric defined above, we also defined the Global Tangling (GTANG) metric obtained by calculating the average of TANG:

$$GTANG = \frac{\sum_{j=1}^{|T|} TANG(t_j)}{|T|}$$

where $|S|$ is the number of source elements in the system.

Table 19. Part of the tangling matrix for a CFVS

		Use cases							
		Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message
Concerns	Insert File	0	0	0	0	0	0	0	0
	Retrieve File	0	1	1	0	0	0	0	0
	Commit File	0	0	0	0	0	0	0	0
	Update Working Files	0	0	0	1	0	0	0	0
	Remove Files	0	0	0	1	0	0	0	0
	Restore File	0	0	0	0	0	0	0	0
	Store Message	0	0	0	0	0	0	0	1
	Retrieve Message	0	0	0	0	0	0	0	1
	Difference	0	0	0	0	0	0	0	0
	Tag a Set of Files	0	1	1	0	0	0	0	0
	Branch a Set of Files	0	0	1	0	0	0	0	0
	Merge Set of Files	0	0	0	1	0	0	0	0
	Persistence	0	0	0	0	0	0	0	1
	Data Representation	0	0	0	0	0	0	0	1
	Synchronization	0	0	0	0	0	0	0	0
	Logging	0	0	0	0	0	0	0	0
	Security	0	1	1	1	0	0	0	1
	Concurrency	0	0	0	0	0	0	0	0

On the other hand, the *Degree of Focus* metric presented in [10] is based on the *Dedication* metric. *Dedication* metric is defined as the relation between the number of lines of code of a component addressing a particular concern and the total number of lines of code of the component. Like we did with the *Concentration* metric, we may represent the *Dedication* one just changing the values of the tangling matrix. In this case, a column in the tangling matrix shows the *Dedication* metric for a component with respect to each concern of the system. Then, having the values of the *Dedication* metric as input, the calculation of the *Degree of Focus* is very simple.

5.3.3 Metrics for crosscutting

In addition to the metrics presented above, the crosscutting product matrix defined in our framework provides very useful information for assessing the degree of crosscutting. This matrix may be used to avoid the problem of having false positives (concerns considered as candidate aspects but they are not really crosscutting concerns). The information shown in a row of the crosscutting product matrix may be interpreted as follows:

- The cell of the diagonal represents the total number of target elements where the source element is crosscutting to other source elements.

- The rest of cells indicate the number of points where the concern of this row is crosscutting to the concern of the corresponding column.

We can see in Table 20 a fragment of the crosscutting product matrix for the Concurrent File Version System mentioned above where this information may be analyzed. The cells corresponding to the diagonal are marked with dark grey background. As an example, we can see that the `Retrieve File` concern is crosscutting to other concerns in three target elements and is crosscutting to the `Tag a Set of Files`, `Branch a Set of Files` and `Security` concerns in two, one and three points respectively. We can also observe in the same table that the `Security` concern crosscuts to other concerns in eleven target elements. Then we may assure that the `Security` concern has a higher degree of crosscutting than the `Retrieve File` one. By means of establishing a threshold value in the diagonal, we can decide whether a concern can be considered as a candidate crosscutting concern or not. Of course, the developer may assist the process deciding the final crosscutting concerns to be taken into account.

Table 20. Part of the crosscutting product matrix for the CFVS

		Concerns																	
		Insert File	Retrieve File	Commit File	Update Working Files	Remove File	Restore File	Store Message	Retrieve Message	Difference	Tag a Set of Files	Branch a Set of Files	Merge Set of Files	Persistence	Data Representation	Synchronization	Logging	Security	Concurrency
	Retrieve File	0	3	0	0	0	0	0	0	0	2	1	0	0	0	0	0	3	0
	Security	1	3	1	2	3	0	2	2	1	2	1	1	3	3	2	0	11	2

Finally we show in Table 21 a summary of the metrics used in this section. Each row in this matrix represents a different metric category while the columns show the concept defined in the metric frameworks compared above corresponding to such a category.

Table 21. Summary of the different metrics

	García et al. [25]	**Aho et al.** [10]	**Crosscutting pattern based framework.**
Metrics for scattering	Concern Diffusion	Degree of Scattering (based on CON)	Rows of the scattering matrix
Metrics for tangling	Component Cohesion	Degree of Focus (based on DEDI)	Columns of the tangling matrix
Metrics for crosscutting	-	-	Diagonal of the crosscutting product matrix
			Rows of the crosscutting product matrix

6. RELATED WORK

Some other publications have addressed the formalization of crosscutting, one of the core concepts in AOSD. As stated in Section 2, our definition of crosscutting is similar to definitions of Masuhara & Kiczales [21], and Tonella and Ceccato [28]. A detailed comparison has been shown in Section 2.

There are also other authors that provided some works on aspect-oriented metrics. On one hand, some of these metrics are focused on assess the degree of scattering of a traditional object-oriented system in order to decide whether an aspect-oriented refactorization is needed [25] [10]. In other works such as [30], the authors try to assess modularity in aspect-oriented systems so that metrics like coupling or

cohesion may be assessed also for aspects. Since we are providing a formal definition of crosscutting, we are interested in metrics for assess modularity in traditional object-oriented systems. Then, in Section 5 we showed how to adapt our framework to the metrics presented in [25] [10] and how to extend them with new ones.

Several authors use matrices (design structure matrices, DSM) to analyze modularity in software design [2]. Lopes and Bajracharya [20] describe a method with clustering and partitioning of the design structure matrix for improving modularity of object-oriented designs. However, the design structure matrices represent intra-level dependencies (as coupling matrices) and not the inter-level dependencies as in the dependency matrices used for our analysis of crosscutting. In [24], a relationship matrix (concern x requirement) is described very similar to our dependency matrix, and used to identify crosscutting concerns. However, there is no explicit definition of crosscutting.

The papers described above lack an application of their definition of crosscutting to consecutive levels. We used our formalization to trace crosscutting concerns through different levels of a software development process, as shown by the cascading operation.

7. CONCLUSION

In this paper, we proposed a conceptual framework for describing crosscutting. We introduced a crosscutting pattern with a mapping between source elements onto target elements. With source and target, we abstract from specific levels or phases in software development. We defined crosscutting, tangling and scattering as separated cases based on specific mappings between source and target. We introduced the dependency matrix and crosscutting matrix to visualize the definitions. We showed that it is possible to formalize these definitions. It is essential to define explicitly the dependency relations used in the mapping between source elements and target elements, as represented in the dependency matrix.

The proposed definitions are similar to definitions of crosscutting in some other publications, e.g. [21], although our definition is not symmetric and less restrictive. We showed a formal comparison of some of these definitions.

An interesting application is the cascading of crosscutting patterns, which can be used to model crosscutting relations across several levels, for example from concern modelling, to requirements, architectural design to detailed design and implementation. As such, it provides an approach for traceability analysis. Another interesting area is the definition of new metrics to assess the degree of crosscutting in a system. As we showed in the paper, the framework allows the representation of some existing metrics and the definition of new ones.

The framework can be applied in concrete cases in order to establish the suitability of the chosen concepts and definitions. The following topics have been investigated in terms of the crosscutting pattern: identification of crosscutting concerns, change impact analysis or integration into a Model Driven Development process. We found important benefits of the application of the framework in those areas. In [4], we performed a detailed analysis on the application of the framework for mining aspects at early phases. The utilization of dependency matrices for this purpose allows the developer to improve traceability of concerns through several refinement levels. In particular, in [4], we performed an analysis of crosscutting across requirements and architecture design phases. In [3], we also performed an analysis of the change impact based on the crosscutting pattern so that we can analyze the change impact in case of crosscutting. Change impact in the traceability pattern is operationalized by means of the elements involved in the change of a source element. Finally, the Model Driven Architecture (MDA) initiative aims at providing stable models amenable to changes [22]. These models can be built at different levels of abstraction (CIM, PIM and PSM) allowing transformations among them. Then an analysis of crosscutting can be made in MDA developments just observing transformation rules performed between

models. In [5] we analyzed in more detail the use of the framework in MDA transformations, e.g. in a case study with the Concurrent File Versioning System.

ACKNOWLEDGEMENT

This work has been carried out in conjunction with the AOSD-Europe Project IST-2-004349-NoE (see [1]) and also partially supported by MEC under contract TIN2005-09405-C02-02. We would like to thank Ana Moreira and Gregor Kiczales for their comments and suggestions and also to Bedir Tekinerdogan for allowing us to use the Concurrent File Version System as an example.

REFERENCES

[1] AOSD-Europe (2005). *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation.* Retrieved May, 2005, from http://www.aosd-europe.net/documents/d9Ont.pdf.

[2] Baldwin, C.Y. & Clark, K.B. (2000). *Design Rules vol I, The Power of Modularity.* MIT Press.

[3] Berg, K. van den (2006). Change Impact Analysis of Crosscutting in Software Architectural Design. In *Workshop on Architecture-Centric Evolution at 20th ECOOP*, Nantes

[4] Berg, K. van den, Conejero, J. M. & Hernández, J. (2006a). Analysis of Crosscutting Across Software Development Phases Based on Traceability. In *Early Aspects Workshop at 28th ICSE*, Shanghai.

[5] Berg, K. van den, Tekinerdogan, B. & Nguyen H. (2006c). Analysis of Crosscutting in Model Transformations. *In J. Aagedal, T. Neple, J. Oldevik (Eds). ECMDA-TW Traceability Workshop Proceedings 2006*. SINTEF Report A219, pp 51-64

[6] Bushmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). Pattern-Oriented Software Architecture: A System of Patterns. Wiley, West Sussex, England, 1996.

[7] Cachero. C., Gómez, J., Párraga, A. & Pastor, O. (2001). Conference Review System: A Case of Study. In [13].

[8] Ceccato M., Marin M., Mens K., Moonen L., Tonella P. and Tourwé T., (2006). Appliying and Combining Three Different Aspect Mining Techniques. Delf University of Technology. Report TUD-SERG-2006-002

[9] Davis, A. (1993). *Software Requirements: Objects, Functions and States.* Prentice–Hall, Second Edition.

[10] Eaddy M., Aho A. (2007). Towards Assessing the Impact of Crosscutting Concerns on Modularity, *AOSD Workshop on Assessment of Aspect Techniques (ASAT 2007),* Vancouver, BC, Canada, March 12, 2007

[11] Ferrante, J., Ottenstein, K.J. and Warren J.D. (1987). The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Language and System, Vol. 9, No. 3*

[12] Filman, R., et al. (2004). *Aspect-Oriented Software Development*. Addison-Wesley

[13] First International Workshop on Web-Oriented Software Technology. (2001). http://www.dsic.upv.es/~west/iwwost01/. Valencia, Spain

[14] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns. Elements of reusable object-oriented software.* Addison-Wesley.

[15] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. and Staa, A. (2005) Modularizing Design Patterns with Aspects: A Quantitative Study. *LNCS Transactions on Aspect-Oriented Software Development*, Springer, 2005.

[16] Kiczales, G. (2005) *Crosscutting*. AOSD.NET Glossary 2005. At http://aosd.net/wiki/index.php?title=Crosscutting.

[17] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. (2001). An overview of AspectJ. *In European Conference on Object-Oriented Programming*, LNCS 2072, Springer, pp.327-353.

[18] Knethen, A. von & Paech, B (2002). A Survey on Tracing Approaches in Practice and Research. IESE-Report No. 095.01/E. v1.0. Fraunhofer Institut Experimentelles Software Engineering.

[19] Laddad, R. (2002). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publicacions Co.

[20] Lopes, C.V. & Bajracharya, S.K. (2005). An analysis of modularity in aspect oriented design. In 4th *International Conference on Aspect-Oriented Software Development*. Chicago, Illinois

[21] Masuhara, H. & Kiczales, G. (2003). Modeling Crosscutting in Aspect-Oriented Mechanisms. In *17th European Conference on Object Oriented Programming*. Darmstadt

[22] MDA (2003). MDA Guide Version 1.0.1, document number omg/2003-06-01

[23] Ramesh, B. & Jarke, M. (2001). Toward reference models for requirements traceability. IEEE Transactions on Software Engineering, 27(4):58–93.

[24] Rashid, A., Moreira, A. & Araujo, J. (2003). Modularisation and Composition of Aspectual Requirements. In *Second Aspect Oriented Software Conference.* Boston, USA.

[25] Sant'Anna, C., Figueiredo, E., Garcia, A., Lucena, C. (2007). On the Modularity Assessment of Software Architectures: Do my architectural concerns count? In *First European Conference on Software Architecture.* Madrid, Spain.

[26] Sutton, S. & Rouvellou, I. (2002). Modeling of Software Concerns in Cosmos. In *First Aspect Oriented Software Development Conference*. Enschede, The Netherlands

[27] Tekinerdogan, B. (2004). ASAAM: Aspectual Software Architecture Analysis Method. In *4th Working IEEE/IFIP Conference on Software Architecture.*

[28] Tonella, P. and Ceccato, M. (2004). Aspect Mining through the Formal Concept Analysis of Execution Traces. In *11th Working Conference on Reverse Engineering*. Delft, the Netherlands.

[29] Zhao, J. and Rinard, M. (2003). System dependence graph construction for aspect-oriented programs. *Technical Report MITLCS-TR-891*, Laboratory for Computer Science, MIT.

[30] Zhao, J. (2004). Measuring Coupling in Aspect-Oriented Systems. Technical report, Information Processing Society of Japan (IPSJ).